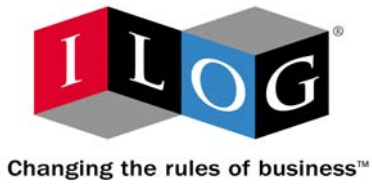

ILOG Solver 6.0

Release Notes

October 2003

Copyright © 1987-2003, ILOG, S. A. — All rights reserved.

Contents

Compatibility with Previous Versions 1
Local Search Library 1
New Class IloIIM 2
Floating-point Library for Non-linear Constraints 2
Changes to License Management 3
Numerical and Integer Division 3
New Default Behavior for IloSolver::setFastRestartMode 3
ILOG Solver Debugger. 3
New Member Function IloSolver::useLinPropagation 4
New Member Function IloSolver::commitSearch 5
Relative Optimization Steps 5
Soft Constraints 5
Native .NET Support for Solver. 6
New Explanation Functionality 6
IloExplainer 6
IloSolverExplainer 7
New Class IlcRandom 8
Constructors. 8
Member Functions 8
New Function IlcPack 9

New Branch Selector Functionality 10
IloBranchSelector 10
IloBranchSelectorI 11
New Environment Variable ILC_NO_MEMORY_MANAGER 11
Known Issues 12
Obsolete Ports 12

ILOG Solver 6.0 Release Notes

These release notes highlight improvements in and changes to ILOG Solver from version 5.3 to version 6.0.

Compatibility with Previous Versions

ILOG Solver 6.0 is compatible with ILOG Solver 5.3. Local search and some floating-point facilities have been separated from the main library. There have been changes made to numerical and integer division in Concert Technology and to the default behavior of `IloSolver::setFastRestartMode`. The following sections detail these changes.

Local Search Library

Solver's local search facilities have been separated from the main library. When Solver is installed, the local search facilities are also installed in the same directory as Solver. These facilities come under the title Iterative Improvement Methods (IIM).

This change requires some changes to makefiles/msvc project files and possibly source code. For a user application, the following transformations must be done in your source code if they exist:

```
#include <ilsolver/local.h>    -> #include <ilsolver/iimls.h>
```

```
#include <ilsolver/ilometa.h>  -> #include <ilsolver/iimmeta.h>
#include <ilsolver/ilonhood.h> -> #include <ilsolver/iimnhood.h>
```

Additionally, you must now link with the IIM library, named `solveriim.lib` for Microsoft compilers and `libsolveriim.a` on UNIX systems. This library may be found in `solver/lib/<PLATFORM>/<PORT>`, where `<PLATFORM>` is a machine/compiler combination, such as ultrasparc32_8_6.2 or msvc6, and `<PORT>` indicates which options were used to compile the library, such as static_pic_mt or stat_mda. This is the same location as for the standard Solver library.

Local search examples are available for Microsoft compilers in file `examplesiim.dsw` (msvc6) or `examplesiim.sln` (msvc.NET) and for UNIX systems via the command `"make executeiim"`.

New Class IloIIM

This version of Solver contains a new class `IloIIM` which takes over the local search API which was previously provided by `IloSolver`.

The following member functions are now obsolete:

```
IloSolver::getNeighborIdentifier
IloSolver::getAtIndex
IloSolver::getAtDelta
```

They have been replaced by:

```
IloIIM::getNeighborIdentifier
IloIIM::getAtIndex
IloIIM::getAtDelt
```

For example, the following code:

```
IloInt i = solver.getAtIndex(nid);
```

would now be written as:

```
IloInt i = IloIIM(solver).getAtIndex(nid);
```

or:

```
IloIIM iim(solver);
iim.getAtIndex(nid);
```

Floating-point Library for Non-linear Constraints

Solver's floating-point facilities have been separated from the main library. This library is distributed with the Solver one in the same directory. This library is needed only if `IloSolver.useNonLinConstraint()` is called.

This change requires some changes to makefiles/msvc project files and possibly source code. For a user application, you must now link with the SolverFloat library, named `solverfloat.lib` for Microsoft compilers and `libsolverfloat.a` on UNIX systems. This library may be found in `lib/<PLATFORM>/<PORT>`, where `<PLATFORM>` is a machine/compiler combination, such as sparc_5_5.0 or msvc6, and `<PORT>` indicates which options were used to compile the library, such as static_pic_mt or stat_mda. The include files are located in `<SOLVERDIR>/include`.

Changes to License Management

In this release of Solver, license management has changed. When there is no license, the program throws an exception instead of aborting. Furthermore, Solver now releases the license when the solver has ended instead of keeping it until the end of the program.

Numerical and Integer Division

There is a new `IloEnv` flag `assumeStrictNumericalDivision`.

When this flag is set to `IloTrue`, all divisions created with the Concert Technology `operator /` are handled as numerical divisions. Integer division is always accessible with the `IloDiv` function. See the Concert Technology documentation for more information on the function `IloDiv`.

If the flag is set to `IloFalse`, divisions created with the Concert Technology `operator /` may be handled as integer or numerical divisions, depending on the case. This use is highly deprecated. The default mode is `IloFalse` for compatibility reasons.

New Default Behavior for IloSolver::setFastRestartMode

The member function `IloSolver::setFastRestartMode` is now the default behavior. This member function controls initial constraint propagation during a search performed by `IloSolver::startNewSearch`. The initial constraint propagation will not be redone if the following conditions are met:

- synchronization mode has been set to `IloSynchronizeAndRestart`
- the model has not changed since the last search
- the parameter mode is `IloTrue`
- there are no objectives in the model

ILOG Solver Debugger

The ILOG Solver Debugger is composed of a GUI and a set of C++ libraries. The C++ libraries enable a C++ application written on top of ILOG Solver or ILOG Scheduler to connect to the Debugger GUI in order to trace and debug its execution or tune its performance. ILOG Solver Debugger contains graphic facilities to monitor the search (Search Tree), the propagation (Propagation Spy and Variable Domains), and to combine search and propagation information (Christmas Tree). It provides a Constraint Profiler to obtain statistics, by constraint, about constraint calls and efficiency in terms of domain reduction. It also enables you to create your own 2D animated graphic representation of the problem being solved. The Solver Debugger is available on Microsoft Windows, Linux, and Solaris ports. For details, refer to the *ILOG Solver 6.0 Debugger User's Manual*.

New Member Function IloSolver::useLinPropagation

This version of Solver contains a new member function `IloSolver::useLinPropagation`. This member function is in beta. After this method is called, all linear constraints added to Solver are no longer handled by the default propagation mechanism of Solver. Instead they are collected by a global propagation mechanism. This global propagation can improve performance in cases where the linear constraints reduce the variable domains only by small increments. An example of its use is given below.

```
int main () {
  IloEnv env;
  IloModel model(env);
  IloSolver solver(model);
  IloIntVar var1(env, 0, 20000000);
  IloIntVar var2(env, 0, 20000000);
  model.add(var1 < var2);
  model.add(var1 > var2);

  solver.useLinPropagation(); // <--------- new code

  if(solver.propagate()) {
    cout << "Propagation succeeded" << endl;
    solver.printInformation();
  }
  else {
    cout << "Propagation failed" << endl;
    solver.printInformation();
  }
  env.end();
  return 0;
}
```

On this example, the use of `useLinePropagation` significantly improves the running time. Note, however, that the aforementioned global propagation mechanism is more costly and hence not always beneficial with regard to the default propagation mechanism of Solver.

New Member Function IloSolver::commitSearch

The following new member function commits the current search to be confined to the part of the search tree below the closest choice point with label `label`. In case no such choice point exists, this method then commits the search to the current state and removes all remaining open choice points or open search nodes. This behavior is the opposite of that of `IloSolver::fail(label)`.

```
void commitSearch(IloAny label)
```

Relative Optimization Steps

This version of Solver contains two new member functions that allow you to implement relative optimization steps.

The following member function returns the relative optimization step. For example, if the optimization step is 90%, Solver searches for a solution where the objective value is less than 90% of the objective value of the best solution found so far.

```
IloSolver::getRelativeOptimizationStep
```

The following member function sets the relative step size used to measure improvement between solutions during a search by `IloSolver::solve`. It is used to update the cutoff each time a mixed integer solution is found. The value is multiplied by the absolute value of the integer objective and subtracted from (added to) the newly found integer objective when minimizing (maximizing). This forces the mixed integer optimization to ignore integer solutions that are not at least this amount better than the one found so far. This value must be strictly between 0 and 1.

```
IloSolver::setRelativeOptimizationStep
```

Soft Constraints

A soft constraint is a constraint that can be violated. Solver provides the user with a mechanism that is able to deal with such constraints.

A soft constraint can be defined only for constraints involving `IlcIntVar` variables.

To define a soft constraint, use the function:

```
IlcSoftConstraint IlcSoftCtHandlerI::createSoftConstraint(IlcConstraint ct);
```

The function `createSoftConstraint` makes a copy of a constraint and copies of all variables on which the constraint is defined.

A soft constraint is added using the function:

```
IlcConstraintI::add(IlcConstraintI* )
```

By adding a soft constraint, a user can simulate what would happen if the constraint had been directly added to the solver without defining a `softConstraint`. The simulation will show the effects of adding the constraint to the solver, but will only modify the copies of the variables, not the original variables. If the constraint from which the `softConstraint` is defined fails in the simulation, no global fail is triggered, instead the member function `whenFail(IlcDemon demon)` is called.

Native .NET Support for Solver

Solver is now delivered with a native .NET port. This Microsoft Visual Studio .NET 2003 library is accessible through C#, VB.NET, and other supported languages. The examples are located in `examples/x86_.net2003_7.1/dotnet` and the library is located in `lib/x86_.net2003_7.1/dotnet`. A .NET API Reference Manual is also available.

New Explanation Functionality

ILOG Solver provides the ability to explain why a particular solution has been proposed. The class `IloExplainer` defines the abstract protocol of this explanation facility. This protocol consists of three queries:

- Why is a constraint `ct` satisfied?
- Why is a constraint `ct` violated?
- Why does a failure (that is, inconsistency) occur?

IloExplainer

Subclasses of the class `IloExplainer` may use different deduction mechanisms for finding explanations. The subclass `IloSolverExplainer` uses the constraint propagation mechanism of Solver as a deduction mechanism.

Constructor

```
IloExplainer(IloExplainerI* impl);
```

This constructor creates a handle of an explanation object with the implementation object `impl`.

Member Functions

```
void end();
```

This member function terminates the explainer and frees internally allocated memory.

```
IloExplainerI* getImpl() const;
```

This member function returns the implementation object of the invoking explanation object.

```
IloModel why(IloConstraint ct);
```

If the constraint `ct` is satisfied in the solution of the invoking explainer, this member function returns an argument for `ct`. The argument is a minimal subset of the set of constraints of this solution that is sufficient to deduce the constraint `ct`. The argument is represented by an instance of the class `IloModel`.

If the constraint `ct` is not satisfied in the solution of the invoking explainer, an exception `IloExplainer::NoExplanationException` is raised.

If `ct` is an `IloExistsComponent` constraint, the argument shows why the associated component was included in the configuration.

```
IloModel whyFail();
```

If the solution of the invoking explainer is inconsistent (that is, it leads to a failure when extracted by a solver), this member function returns a conflict. The conflict is a minimal subset of the set of constraints of this solution that is sufficient to produce a failure. The conflict is represented by an instance of the class `IloModel`.

If the solution of the invoking explainer is consistent, an exception `IloExplainer::NoExplanationException` is raised.

```
IloModel whyNot(IloConstraint ct);
```

If the constraint `ct` is violated in the solution of the invoking explainer, this member function returns a counterargument for `ct`. The counterargument is a minimal subset of the set of constraints of this solution that is sufficient to refute the constraint `ct`. The counterargument is represented by an instance of the class `IloModel`.

If the constraint `ct` is not violated in the solution of the invoking explainer, an exception `IloExplainer::NoExplanationException` is raised.

IloSolverExplainer

ILOG Solver now provides the ability to explain why a particular solution has been proposed. An instance of the class `IloSolverExplainer` is an explainer that uses the propagation mechanism of an `IloSolver` for deducing queries and fails.

Constructors

```
IloSolverExplainer(IloSolverExplainerI* impl=0);
```

This constructor creates a handle of a Solver explainer with the implementation object `impl`. The class `IloSolverExplainerI` is not documented.

```
IloSolverExplainer(IloModel model);
```

This constructor creates a Solver explainer for the model `model`. The explanations found by this explainer are subsets of the set of constraints of `model`.

New Class IlcRandom

Objects of this class produce streams of pseudo-random numbers. You can use objects of this class to create a search with a random element. This class will produce the same stream of random numbers in recompute mode as when it was in compute mode.

Constructors

```
IlcRandom()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
IlcRandom(IlcRandomI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
IlcRandom(IloSolver m, IlcInt seed)
```

This constructor creates a random number generator, initially seeded with the seed `seed`.

Member Functions

```
void copy(IlcRandom rnd)
```

This member function copies the state of another random number generator to the invoking one. After the copy, both generators will produce the same stream of pseudo-random numbers.

```
void copyTo(IloRandom rnd)
```

This member function copies the state of the invoking generator to an ILOG Concert Technology random number generator. After the copy, both generators will produce the same stream of pseudo-random numbers.

```
IlcFloat getFloat()
```

This member function returns a floating point number drawn uniformly from the range `[0..1)`.

```
IlcRandomI * getImpl()
```

This member function returns a pointer to the implementation object of the invoking handle.

```
IlcInt getInt(IlcInt n)
```

This member function returns a integer number drawn uniformly from the range `[0..n)`.

```
const char * getName()
```

This member function returns the name of the invoking object.

```
IloAny getObject()
```

This member function returns a pointer to the external object associated with the invoking object, if there is such an association. It returns 0 (zero) otherwise.

```
IloSolver getSolver()
```

This member function returns the solver (a handle) of the invoking object.

```
void operator=(const IlcRandom & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

```
void reSeed(IlcInt seed)
```

This member function reseeds the generator with seed `seed`.

```
void setName(const char * name)
```

This member function sets the name of the invoking object to a copy of name. This assignment is a reversible action.

```
void setObject(IloAny object)
```

This member function establishes a link between the invoking object and an external object of which the invoking object might be a data member.

New Function IlcPack

The `IlcPack` function returns a constraint which maintains the load of a set of containers, given a set of weighted items and an assignment of items to containers.

Consider that you have *n* items and *m* containers. Each item *i* has an integer weight *w[i]* and a constrained integer variable *p[i]* associated with it indicating in which container (numbered contiguously from 0) item *i* is to be placed. No item can be split up, and so can go in only one container. Also associated with each container *j* is an integer variable *l[j]* representing the load in that container; that is, the sum of the weights of the items which have been assigned to that container. A capacity can be set for each container placing an

upper bound on this load variable. The constraint also ensures that the total sum of the loads of the containers is equal to the sum of the weights of the items being placed.

```
IlcConstraint IlcPack(IlcIntVarArray load,
                      IlcIntVarArray where,
                      IlcIntArray weight);
IlcConstraint IlcPack(IlcIntVarArray load,
                      IlcIntVarArray where,
                      IlcInt weight = 1);
```

New Branch Selector Functionality

The class `IloBranchSelector` represents branch selectors in a Concert Technology model. The class `IlcBranchSelector` represents branch selectors internally in a Solver search.

A branch selector is used to shape the explored part of the search tree.

IloBranchSelector

An instance of this class represents a branch selector in a Concert Technology model. Branch selectors are useful in goals (such as the goal returned by `IloApply` or other instances of `IloGoal`) to shape the exploration of the search tree during the search for a solution.

Constructors

```
IloBranchSelector()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
IloBranchSelector(IloBranchSelectorI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

Member Functions

```
IloBranchSelectorI * getImpl()
```

This member function returns a pointer to the implementation object of the invoking handle.

```
void operator=(const IloBranchSelector & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

IloBranchSelectorI

This class is the implementation class for `IloBranchSelector`.

Destructor

```
~IloBranchSelectorI()
```

This destructor is called automatically by the destructor of its subclasses. It frees memory used by the invoking object.

Member Functions

```
virtual void display(ostream & )
```

This member function prints the invoking branch selector on an output stream.

```
virtual IlcBranchSelector extract(const IloSolver solver)
```

In general terms, in Concert Technology, the objects of a model must be extracted for an algorithm (an instance of one of the subclasses of `IloAlgorithm`, such as `IloSolver`). This member function returns the internal branch selector extracted for Solver from the invoking branch selector of a model.

```
virtual IloBranchSelectorI * makeClone(IloEnvI * env)
```

This member function is called internally to duplicate the current branch selector.

New Environment Variable ILC_NO_MEMORY_MANAGER

This operating-system environment variable enables you to control the memory manager of Solver. It replaces `IloSolver::useHeap`.

Solver uses its own memory manager to provide faster memory allocation for certain Solver objects. The use of this memory manager can hide memory problems normally detected by memory usage applications (such as Rational Purify, for example). If you are working in a software development environment capable of detecting bad memory access, you can use this operating-system environment variable to turn off the Solver memory manager in order to detect such anomalies during software development. This environment variable also applies to reversible memory.

For example, if you are working in such a development environment on a personal computer running Microsoft NT, use this statement:

```
set ILC_NO_MEMORY_MANAGER=1
```

If you are working on a UNIX platform, using a C-shell, here is one way of setting this environment variable:

```
setenv  ILC_NO_MEMORY_MANAGER
```

Known Issues

There is a known issue involving a crash on Microsoft Windows when mixing libraries. All Solver libraries are compiled in non-debug mode. If you mix debug and non-debug libraries when compiling examples or applications using Microsoft Windows, you may experience a crash. This issue is described in Microsoft Knowledge Base Article 140584 at http://support.microsoft.com/default.aspx?scid=http://support.microsoft.com:80/support/kb/articles/q140/5/84.asp&NoWebContent=1.

There is a known issue involving an unavoidable warning on alpha platforms. The compilation of each file generates the following message:

```
cxx: Warning: IEEE dynamic rounding mode is not supported for long doubles on
this platform.
```

ILOG Solver does not use long double in its source code. Therefore, the warning is not relevant. Unfortunately, it is unavoidable.

There is a known issue involving an unavoidable warning on Linux platforms. For example, the compilation produces warning messages like the following:

```
../../../examples/src/crews.cpp:124: warning: cannot pass objects of non-POD
type "class IloIntSetVar" through ...
```

This is a GCC bug as handle classes, which are POD objects, are not recognized as such. This warning is also unavoidable.

Obsolete Ports

With this release of ILOG Solver, certain ports have been removed from the distribution. For most of these ports, more recent compiler and/or operating system versions are now supported. Below is a list of the removed ports and the recommended replacement ports. ILOG has a port policy which can be found on the ILOG support web site for your region.

Table 1 *Solver 6.0 Port Support Changes*

Platform	Dropped Support	Recommended Replacement
alpha	alpha_4_6.2/static	alpha_5.1_6.5/static
HP	hp32_11_3.15/static_pic_mt hp32_pa8000_3.15/static_pic_mt hp64_11_3.15/static_mt	hp32_11_3.39/static_pic_mt static_pic_mt_AA (STL port) (The hp32 ports default to PA8000 architecture.) hp64_11_3.39/static_pic_mt static_pic_mt_AA (STL port)
Linux	i86_linux2_glibc2.2_gcc3.0/ static_mt static_pic_mt	i86_linux2_glibc2.3_gcc3.2/ static_pic
Solaris	sparc_5_5.0/static_pic_mt ultrasparc_5_5.0/static_pic_mt ultrasparc64_5_5.0/static_pic_mt	ultrasparc32_8_6.2/static_pic_mt ultrasparc64_8_6.2/static_pic_mt
RS6000	rs6000_4.3_5.0/static_mt rs6000_4.3_5.0/static_mt_stl rs6000_4.3_5.0/static_stl rs64_4.3_5.0/static rs64_4.3_5.0/static_mt rs64_5.1_5.0/static rs64_5.1_5.0/static_mt	rs6000_5.1_6.0/static_stl rs6000_5.1_6.0/static_mt_stl rs64_5.1_6.0/static_stl rs64_5.1_6.0/static_mt_stl
Windows	ia64_win64_vc6/stat_sta, stat_mta, stat_mda msvc6/stat_st, stat_mt, stat_md, stat_sta, stat_mta	ia64_.net2003_7.1/stat_mda x86_.net2003_7.1/stat_sta, stat_mta, stat_mda (These new ports correspond to the Microsoft Visual Studio .NET 2003 C++ compiler.)

Note: *Please note that Microsoft Corporation has announced plans to discontinue standard support for the Microsoft MSVC6 compiler as of September 30, 2003. Accordingly, ILOG will no longer be able to provide standard support for those ILOG customers that have not upgraded their applications to a later version of the Microsoft compiler. Nevertheless, ILOG will continue to provide bug fixes for these customers, to the extent ILOG can do so without the support of Microsoft.*